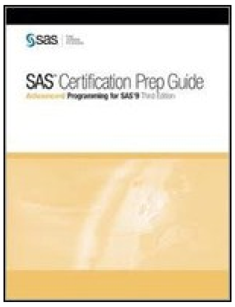


SAS Certification Prep Guide: Advanced Programming for SAS 9, Third Edition

by SAS Institute

Chapter 2: Performing Advanced Queries Using PROC SQL

Overview

Introduction

The SELECT statement is the primary tool of PROC SQL. Using the SELECT statement, you can identify, manipulate, and retrieve columns of data from one or more tables and views.

You should already know how to create basic PROC SQL queries by using the SELECT statement and most of its subordinate clauses. To build on your existing skills, this chapter presents a variety of useful query techniques, such as the use of subqueries to subset data.

The PROC SQL query shown below illustrates some of the new query techniques that you will learn:

```
proc sql outobs=20;
title 'Job Groups with Average Salary';
title2 '> Company Average';
   select jobcode,
          avg(salary) as AvgSalary format=dollar11.2,
          count(*) as Count
      from sasuser.payrollmaster
      group by jobcode
      having avg(salary) >
         (select avg(salary)
            from sasuser.payrollmaster)
order by avgsalary desc;
```

Job Groups with Average Salary > Company Average		
JobCode	**AvgSalary**	**Count**
PT3	$154,706.50	2
PT2	$122,253.40	10
PT1	$95,071.13	8
NA2	$73,336.00	3
ME3	$59,374.86	7
NA1	$58,845.00	5
TA3	$55,551.42	12

Objectives

In this chapter, you learn to

- display all rows, eliminate duplicate rows, and limit the number of rows displayed
- subset rows using other conditional operators and calculated values
- enhance the formatting of query output
- use summary functions, such as COUNT, with and without grouping
- subset groups of data by using the HAVING clause
- subset data by using correlated and noncorrelated subqueries
- validate query syntax.

Prerequisites

Before you begin this chapter, you should complete the following chapter:

- "Performing Queries Using PROC SQL" on page 4.

Viewing SELECT Statement Syntax

The SELECT statement and its subordinate clauses are the building blocks for constructing all PROC SQL queries.

General form, SELECT statement:

```
SELECT column-l<, … column-n>
     FROM table-1 | view-l<, … table-n | view-n>
     <WHERE expression>
     <GROUP BY column-l<, … column-n>>
     <HAVING expression>
     <ORDER BY column-l<, … column-n>>;
```

where

SELECT

specifies the column(s) that will appear in the output

FROM

specifies the table(s) or view(s) to be queried

WHERE

uses an expression to subset or restrict the data based on one or more condition(s)

GROUP BY

classifies the data into groups based on the specified column(s)

HAVING

uses an expression to subset or restrict groups of data based on group condition(s)

ORDER BY

sorts the rows that the query returns by the value(s) of the specified column(s).

Note The clauses in a PROC SQL SELECT statement *must* be specified in the order shown.

You should be familiar with all of the SELECT statement clauses except for the HAVING clause. The use of the HAVING clause is presented later in this chapter.

Now, we will look at some ways you can limit and subset the number of columns that will be displayed in query output.

Displaying All Columns

You already know how to select specific columns for output by listing them in the SELECT statement. However, for some tasks, you will find it useful to display *all* columns of a table concurrently. For example, before you create a complex query, you might want to see the contents of the table you are working with.

Using SELECT *

To display all columns in the order in which they are stored in a table, use an asterisk (*) in the SELECT clause. All rows will also be displayed, by default, unless you limit or subset them.

The following SELECT step displays all columns and rows in the table *Sasuser.Staffchanges*, which lists all employees in a company who have had changes in their employment status.

```
proc sql;
   select *
```

```
      from sasuser.staffchanges;
```

As shown in the output, the table contains six columns and six rows.

EmpID	LastName	FirstName	City	State	PftoneN umber
1639	CARTER	KAREN	STAMFORD	CT	203/781-8839
1065	CHAPMAN	NEIL	NEW YORK	NY	718/384-5618
1561	SANDERS	RAYMOND	NEW YORK	NY	212/588-6615
1221	WALTERS	DIANE	NEW YORK	NY	718/384-1918
1447	BRIDESTON	AMY	NEW YORK	NY	718/384-1213
1998	POWELL	JIM	NEW YORK	NY	718/384-8642

Using the FEEDBACK Option

When you specify SELECT*, you can also use the *FEEDBACK* option in the PROC SQL statement, which writes the expanded list of columns to the SAS log. For example, the PROC SQL query shown below contains the FEEDBACK option:

```
proc sql feedback;
   select *
      from sasuser.staffchanges;
```

This query produces the following feedback in the SAS log.

Table 2.1: SAS Log

```
202 proc sql feedback;
203 select *
204 from sasuser.staffchanges;
NOTE: Statement transforms to:

      select STAFFCHANGES.EmpID,
STAFFCHANGES.LastName, STAFFCHANGES.FirstName,
STAFFCHANGES.City, STAFFCHANGES.State,
STAFFCHANGES.PhoneNumber
          from SASUSER.STAFFCHANGES
```

The FEEDBACK option is a debugging tool that lets you see exactly what is being submitted to the SQL processor. The resulting message in the SAS log not only expands asterisks (*) into column lists, but it also resolves macro variables and places parentheses around expressions to show their order of evaluation.

Limiting the Number of Rows Displayed

Overview

When you create PROC SQL queries, you will sometimes find it useful to limit the number of rows that PROC SQL displays in the output. To indicate the maximum number of rows to be displayed, you can use the *OUTOBS=* option in the PROC SQL statement.

General form, PROC SQL statement with OUTOBS= option:

PROC SQL OUTOBS= *n*;

where

n

specifies the number of rows.

Note The OUTOBS= option restricts the rows that are *displayed*, but not the rows that are *read*. To restrict the number of rows that PROC SQL takes as input from any single source, use the *INOBS*= option. For more information about the INOBS= option, see "Managing Processing Using PROC SQL" on page 278.

Example

Suppose you want to quickly review the types of values that are stored in a table, without printing out all the rows. The following PROC SQL query selects data from the table *Sasuser.Flightschedule*, which contains over 200 rows. To print only the first 10 rows of output, you add the OUTOBS= option to the PROC SQL statement.

```
proc sql outobs=10;
   select flightnumber, date
      from sasuser.flightschedule;
```

FlightNumber	Date
132	01MAR2000
132	01MAR2000
132	01MAR2000
132	01 MAR2000
132	01MAR2000
132	01MAR2000
182	01MAR2000
182	01MAR2000
182	01MAR2000
182	01MAR2000

When you limit the number of rows that are displayed, a message similar to the following appears in the SAS log.

Table 2.2: SAS Log

```
WARNING: Statement terminated early due to OUTOBS=10 option.
```

Note The OUTOBS= and INOBS= options will affect tables that are created by using the CREATE TABLE statement and your report output.

Note In many of the examples in this chapter, OUTOBS= is used to limit the number of rows that are displayed in output.

Eliminating Duplicate Rows from Output

In some situations, you might want to display only the *unique* values or combinations of values in the column(s) listed in the SELECT clause. You can eliminate duplicate rows from your query results by using the keyword *DISTINCT* in the SELECT clause. The DISTINCT keyword applies to *all* columns, and *only* those columns, that are listed in the SELECT clause. We will see how this works in the following example.

Example

Suppose you want to display a list of the unique flight numbers and destinations of all international flights that are flown during the month.

The following SELECT statement in PROC SQL selects the columns **`FlightNumber`** and **`Destination`** in the table *Sasuser.Internationalflights*:

```
proc sql outobs=12;
   select flightnumber, destination
      from sasuser.internationalflights;
```

Here is the output.

FlightNumber	Destination
182	YYZ
219	LHR
387	CPH
622	FRA
821	LHR
132	YYZ
271	CDG
182	YYZ
219	LHR
387	CPH
622	FRA
821	LHR

As you can see, there are several duplicate pairs of values for **FlightNumber** and **Destination** in the first 12 rows alone. For example, flight number 182 to YYZ appears in rows 1 and 8. The entire table contains many more rows with duplicate values for each flight number and destination because each flight has a regular schedule.

To remove rows that contain duplicate values, add the keyword DISTINCT to the SELECT statement, following the keyword SELECT, as shown in the following example:

```
proc sql;
   select distinct flightnumber, destination
      from sasuser.internationalflights
      order by 1;
```

With duplicate values removed, the output will contain many fewer rows, so the OUTOBS= option has been removed from the PROC SQL statement. Also, to sort the output by **FlightNumber** (column 1 in the SELECT clause list), the ORDER BY clause has been added.

Here is the output from the modified program.

FlightNumber	Destination
132	YYZ
182	YYZ
219	LHR
271	CDG
387	CPH
622	FRA
821	LHR

There are *no duplicate rows* in the output. There are seven unique **FlightNumber-Destination** value pairs in this table.

Subsetting Rows by Using Conditional Operators

Overview

In the WHERE clause of a PROC SQL query, you can specify any valid SAS expression to subset or restrict the data that is displayed in output. The expression might contain any of various types of operators, such as the following.

Type of Operator	Example

comparison	`where membertype ='GOLD'`
logical	`where visits<=3 or status= 'new'`
concatenation	`where name=trim(last) ||', ' ||first`

Note For a complete list of operators that can be used in SAS expressions, see the SAS documentation.

Using Operators in PROC SQL

Comparison, logical, and concatenation operators are used in PROC SQL as they are used in other SAS procedures. For example, the following WHERE clause contains

- the logical operator *AND*, which joins multiple conditions
- two comparison operators: an equal sign (=) and a greater than symbol (>).

```
proc sql;
   select ffid, name, state, pointsused
      from sasuser.frequentflyers
      where membertype='GOLD' and pointsused>0
      order by pointsused;
```

In PROC SQL queries, you can also use the following *conditional operators*. All of these operators except for ANY, ALL, and EXISTS, can also be used in other SAS procedures.

Conditional Operator	Tests for ...	Example
BETWEEN-AND	values that occur within an inclusive range	`where salary between 70000 and 80000`

CONTAINS or ?	values that contain a specified string	`where name contains 'ER'`
		`where name ? 'ER'`

IN	values that match one of a list of values	`where code in ('PT' , 'NA', 'FA')`

IS MISSING or IS NULL	missing values	`where dateofbirth is missing`
		`where dateofbirth is null`

LIKE (with %, _)	values that match a specified pattern	`where address like '% P%PLACE'`
=*	values that *sound like* a specified value	`where lastname =* 'Smith'`
ANY	values that meet a specified condition with respect to *any one* of the values returned by a subquery	`where dateofbirth < any (select dateofbirth from sasuser.payrollmaster where jobcode='FA3')`
ALL	values that meet a specified condition with respect to *all* the values returned by a subquery	`where dateofbirth < all (select dateofbirth from sasuser.payrollmaster where jobcode='FA3')`
EXISTS	the existence of values returned by a subquery	`where exists (select * from sasuser.flightschedule where fa.empid= flightschedule.empid)`

Tip To create a negative condition, you can precede any of these conditional operators, except for ANY and ALL, with the *NOT* operator.

Most of these conditional operators, and their uses, are covered in the next several sections. ANY, ALL, and EXISTS are discussed later in the chapter.

Using the BETWEEN-AND Operator to Select within a Range of Values

To select rows based on a range of numeric or character values, you use the *BETWEEN-AND* operator in the WHERE clause. The BETWEEN-AND operator is inclusive, so the values that you specify as limits for the range of values are included in the query results, in addition to any values that occur between the limits.

General form, BETWEEN-AND operator:

BETWEEN *value-1* **AND** *value-2*

where

value-1

is the value at the one end of the range

value-2

is the value at the other end of the range.

Note When specifying the limits for the range of values, it is *not* necessary to specify the smaller value first.

Here are several examples of WHERE clauses that contain the BETWEEN-AND operator. The last example shows the use of the NOT operator with the BETWEEN-AND operator.

Example	Returns rows In which…
`where date between '01mar2000'd and '07mar2000'd`	the value of **`Date`** is *01mar2000, 07mar2000*, or any date value in between
In this example, the values are specified as date constants.	
`where salary between 70000 and 80000`	the value of **`Salary`** is *70000, 80000*, or any numeric value in between
`where salary not between 70000 and 80000`	the value of **`Salary`** is *not* between or equal to *70000* and *80000*

Using the CONTAINS or Question Mark (?) Operator to Select a String

The *CONTAINS* or question mark (?) operator is usually used to select rows for which a character column includes a particular string. These operators are interchangeable.

General form, CONTAINS operator:

sql-expression **CONTAINS** *sql-expression*

sql-expression ? sql-expression

where

sql-expression

is a character column, string (character constant), or expression. A string is a sequence of characters to be matched that must be enclosed in quotation marks.

Note PROC SQL retrieves a row for output no matter where the string (or second sqlexpression) occurs within the column's (or first sqlexpression's) values. Matching is case sensitive when making comparisons.

Note The CONTAINS or question mark (?) operator is not part of the ANSI standard; it is a SAS enhancement.

Example

The following PROC SQL query uses CONTAINS to select rows in which the **`Name`** column contains the string *ER*. As the output shows, all rows that contain *ER* anywhere within the **`Name`** column are displayed.

```
proc sql outobs=10;
   select name
    from sasuser.frequentflyers
    where name contains 'ER';
```

Name
COOPER, LESLIE
COOPER, ANTHONY
COOK, JENNIFER
FOSTER, GERALD
BRADLEY, JEREMY
BURKE, CHRISTOPHER
AVERY, JERRY
EDGERTON, JOSHUA
SAYERS, RANDY
WANG, CHRISTOPHER

Using the IN Operator to Select Values from a List

To select only the rows that match one of the values in a list of fixed values, either numeric or character, use the IN operator.

General form, IN operator:

column **IN** *(constant-l<,…constant-n>)*

where

column

specifies the selected column name

constant-1 and *constant-n*

represent a list that contains one or more specific values. The list of values must be enclosed in parentheses and separated by either commas or spaces. Values can be either numeric or character. Character values must be enclosed in quotation marks.

Here are examples of WHERE clauses that contain the IN operator.

Example	Returns rows in which…
`where jobcategory in ('PT','NA','FA')`	the value of **`JobCategory`** is *PT, NA*, or *FA*
`where dayof Week in (2,4,6)`	the value of **`Dayof Week`** is 2, 4, or 6

`where chesspiece not in ('pawn','king','queen')`	the value of **`chesspiece`** is *rook, knight*, or *bishop*

Using the IS MISSING or IS NULL Operator to Select Missing Values

To select rows that contain missing values, both character and numeric, use the *IS MISSING* or *ISNULL* operator. These operators are interchangeable.

General form, IS MISSING or IS NULL operator:

column **IS MISSING**

column **IS NULL**

where

column

specifies the selected column name.

Note The IS MISSING operator is not part of the ANSI standard for SQL. It is a SAS enhancement.

Example

Suppose you want to find out whether the table *Sasuser. March/lights* has any missing values in the column **`Boarded`**. You can use the following PROC SQL query to retrieve rows from the table that have missing values:

```
proc sql;
   select boarded, transferred,
          nonrevenue, deplaned
      from sasuser.marchflights
      where boarded is missing;
```

The output shows that two rows in the table have missing values for **`Boarded`**.

Boarded	Transferred	NonRevenue	Deplaned
.	9	0	210
.	16	5	79

Tip Alternatively, you can specify missing values without using the IS MISSING or IS NULL operator, as shown in the following examples:

```
where boarded = .
where flight = ' '
```

However, the advantage of using the IS MISSING or IS NULL operator is that you do not have to specify the data type (character or numeric) of the column.

Using the LIKE Operator to Select a Pattern

To select rows that have values that match as specific *pattern* of characters rather than a fixed character string, use the *LIKE* operator. For example, using the LIKE operator, you can select all rows in which the **`LastName`** value starts with H. (If you wanted to select all rows in which the last name contains the string *HAR*, you would use the CONTAINS operator.)

General form, LIKE operator:

column **LIKE** *'pattern'*

where

column

specifies the column name

pattern

specifies the pattern to be matched and contains one or both of the special characters underscore (_) and percent sign (%). The entire pattern must be enclosed in quotation marks and matching is case sensitive.

When you use the LIKE operator in a query, PROC SQL uses pattern matching to compare each value in the specified column with the pattern that you specify using the LIKE operator in the WHERE clause. The query output displays all rows in which there is a match.

You specify a pattern using one or both of the special characters shown below.

Special Character	Represents
underscore(_)	any single character
percent sign (%)	any sequence of zero or more characters

Note The underscore (_) and percent sign (%) are sometimes referred to as wildcard characters.

Specifying a Pattern

To specify a pattern, combine one or both of the special characters with any other characters that you want to match. The special characters can appear before, after, or on both sides of other characters.

Consider how the special characters can be combined to specify a pattern. Suppose you are working with a table column that contains the following list of names:

- Diana
- Diane
- Dianna
- Dianthus
- Dyan

Here are several patterns that you can use to select one or more of the names from the list. Each pattern uses one or both of the special characters.

LIKE Pattern	Name(s) Selected
LIKE 'D_an'	Dyan
LIKE 'D_an_'	Diana, Diane
LIKE "D_an_	Dianna
LIKE 'D_an%'	all names from the list

Example

The following PROC SQL query uses the LIKE operator to find all frequent-flyer club members whose street name begins with *P* and ends with the word *PLACE*. The following PROC SQL step performs this query:

```
proc sql;
   select ffid, name, address
      from sasuser.frequentflyers
      where address like '% P%PLACE';
```

The pattern '% *P%PLACE*' specifies the following sequence:

- any number of characters (%)
- a space
- the letter *P*
- any number of characters (%)
- the word *PLACE*.

Here are the results of this query.

FFID	Name	Address
WD8375	COOPER. ANTHONY	12 PIEDPIPER PLACE
WD6271	MORGAN. GEORGE	39 PEPPER PLACE
WD6184	STARR. WILLIAM	12 PINEY PLACE
WD2118	JOHNSON. ANTHONY	78 PIPER PLACE
WD3827	KING. WILLIAM	14 PICTURE PLACE
WD8789	HOWARD. LEONARD	45 PECAN PLACE
WD6169	WILDER, NEIL	78 PUMPKIN PLACE
WD8667	YOUNG. DEBORAH	53 PINE PLACE
WD5687	EDWARDS. JENNIFER	3 PEGBOARD PLACE

Using the Sounds-Like (=*) Operator to Select a Spelling Variation

To select rows that contain a value that sounds like another value that you specify, use the sounds-like operator (=*) in the WHERE clause.

General form, sounds-like (=*)operator:

sql-expression =* *sql-expression*

where

sql-expression

is a character column, string (character constant), or expression. A string is a sequence of characters to be matched that must be enclosed in quotation marks.

The sounds-like (=*) operator uses the SOUNDEX algorithm to compare each value of a column (or other sql-expression) with the word or words (or other sql-expression) that you specify. Any rows that contain a spelling variation of the value that you specified are selected for output.

For example, here is a WHERE clause that contains the sounds-like operator:

```
where lastname =* 'Smith';
```

The sounds-like operator does not always select all possible values. For example, suppose you use the preceding WHERE clause to select rows from the following list of names that sound like Smith:

- Schmitt
- Smith
- Smithson
- Smitt

- Smythe

Two of the names in this list will *not* be selected: *Schmitt* and Smithson.

> **Note** The SOUNDEX algorithm is English-biased and is less useful for languages other than English. For more information about the SOUNDEX algorithm, see the SAS documentation.

Subsetting Rows by Using Calculated Values

Understanding How PROC SQL Processes Calculated Columns

You should already know how to define a new column by using the SELECT clause and performing a calculation. For example, the following PROC SQL query creates the new column **`Total`** by adding the values of three existing columns: **`Boarded, Transferred`**, and **`Nonrevenue:`**

```
proc sql outobs=10;
   select flightnumber, date, destination,
          boarded + transferred + nonrevenue
          as Total
   from sasuser.marchflights
```

You can also use a calculated column in the WHERE clause to subset rows. However, because of the way in which SQL queries are processed, you cannot just specify the column alias in the WHERE clause. To see what happens, we will take the preceding PROC SQL query and add a WHERE clause in the SELECT statement to reference the calculated column **`Total`**, as shown below:

```
proc sql outobs=10;
   select flightnumber, date, destination,
          boarded + transferred + nonrevenue
          as Total
      from sasuser.marchflights
      where total < 100;
```

When this query is executed, the following error message is displayed in the SAS log.

Table 2.3: SAS Log

```
519  proc sql outobs=10;
520       select flightnumber, date, destination,
521              boarded + transferred + nonrevenue
522              as Total
523          from sasuser.marchflights
524          where total < 100;
ERROR: The following columns were not found in the contributing tables: total.
```

This error message is generated because, in SQL queries, the WHERE clause is processed before the SELECT clause. The SQL processor looks in the table for each column named in the WHERE clause. The table *Sasuser.Marchflights* does not contain a column named **`Total`**, so SAS generates an error message.

Using the Keyword CALCULATED

When you use a column alias in the WHERE clause to refer to a calculated value, you must use the keyword *CALCULATED* along with the alias. The CALCULATED keyword informs PROC SQL that the value is calculated within the query. Now, the PROC SQL query looks like this:

```
proc sql outobs=10;
   select flightnumber, date, destination,
          boarded + transferred + nonrevenue
          as Total
      from sasuser.marchflights
      where calculated total < 100;
```

This query executes successfully and produces the following output.

FlightNumber	Date	Destination	Total

982	01MAR2000	DFW	70
416	01MAR2000	WAS	93
829	01MAR2000	WAS	96
416	02MAR2000	WAS	90
302	02MAR2000	WAS	93
132	03MAR2000	YYZ	88
921	03MAR2000	DFW	85
290	05MAR2000	WAS	55
523	05MAR2000	ORD	59
416	05MAR2000	WAS	31

Note As an alternative to using the keyword CALCULATED, repeat the calculation in the WHERE clause. However, this method is inefficient because PROC SQL has to perform the calculation twice. In the preceding query, the alternate WHERE statement would be:

```
where 1boarded + transferred + nonrevenue <100;
```

You can also use the CALCULATED keyword in other parts of a query. In the following example, the SELECT clause calculates the new column **`Total`** and then calculates a second new column based on **`Total`**. To create the second calculated column, you have to specify the keyword CALCULATED in the SELECT clause.

```
proc sql outobs=10;
   select flightnumber, date, destination,
          boarded + transferred + nonrevenue
          as Total,
          calculated total/2 as Half
   from sasuser.marchflights;
```

This query produces the following output.

FlightNumber	Date	Destination	Total	Half
182	01MAR2000	YYZ	123	61.5
114	01MAR2000	LAX	196	98
202	01MAR2000	ORD	167	83.5
219	01MAR2000	LHR	222	111
439	01MAR2000	LAX	185	92.5
387	01MAR2000	CPH	163	81.5
290	01MAR2000	WAS	119	59.5
523	01MAR2000	ORD	200	100
982	01MAR2000	DFW	70	35
622	01MAR2000	FRA	227	113.5

Note The CALCULATED keyword is a SAS enhancement and is not specified in the ANSI Standard for SQL.

Enhancing Query Output

Overview

When you are using PROC SQL, you might find that the data in a table is not formatted as you would like it to appear. Fortunately, with PROC SQL you can use enhancements, such as the following, to improve the appearance of your query output:

- column labels and formats
- titles and footnotes
- columns that contain a character constant.

You know how to use the first two enhancements with other SAS procedures. You can also enhance PROC SQL query output by working with the following query:

```
proc sql outobs=15;
   select empid, jobcode, salary,
          salary * .10 as Bonus
      from sasuser.payrollmaster
      where salary>75000
      order by salary desc;
```

This query limits output tol5 observations. The SELECT clause selects three existing columns from the table *Sasuser.payrollmaster*, and calculates a fourth **(`Bonus`)**. The WHERE clause retrieves only rows in which salary is greater than 75,000. The ORDER BY clause sorts by the **`Salary`** column and uses the keyword DESC to sort in descending order.

Here is the output from this query.

EmpID	JobCode	Salary	Bonus
1118	PT3	$155,931	15593.1
1777	PT3	$153,482	15348.2
1404	PT2	$127,926	12792.6
1107	PT2	$125,968	12596.8
1928	PT2	$125,801	12580.1
1106	PT2	$125,485	12548.5
1333	PT2	$124,048	12404.8
1890	PT2	$120,254	12025.4
1410	PT2	$118,559	11855.9
1442	PT2	$118,350	11835
1830	PT2	$118,259	11825.9
1478	PT2	$117,884	11788.4
1556	PT1	$99,889	9988.9
1439	PT1	$99,030	9903
1428	PT1	$95,274	9527.4

Note The **`Salary`** column has the format DOLLAR9. specified in the table.

Look closely at this output and you will see that improvements can be made. You will learn how to enhance this output in the following ways:

- replace original column names with new labels
- specify a format for the **`Bonus`** column, so that all values are displayed with the same number of decimal places
- display a title at the top of the output
- add a column using a character constant.

Specifying Column Formats and Labels

By default, PROC SQL formats output using column attributes that are already saved in the table or, if none are saved, the default attributes. To control the formatting of columns in output, you can specify column modifiers, such as LABEL= and FORMAT^, after any column name specified in the SELECT clause. When you define a new column in the SELECT clause, you can assign a label rather than an alias, if you prefer.

Column Modifier	Specifies…	Example

LABEL=	the label to be displayed for the column	`select hiredate`
		`       label='Date of Hire'`

1ORMAI	the fonnat used to display column data	`select hiredate format=date9.`

Note LABEL= and FORMAT= are not part of the ANSI standard. These column modifiers are SAS enhancements.

Tip To force PROC SQL to ignore permanent labels in a table, specify the NOLABEL system option.

Your first task is to specify column labels for the first two columns. Below, the *LABEL=* option has been added after both **EmpID** and **JobCode**, and the text of each label is enclosed in quotation marks. For easier reading, each of the four columns in the SELECT clause is now listed on its own line.

```
proc sql outobs=15;
   select empid label='Employee ID',
          jobcode label='Job Code',
          salary,
          salary * .10 as Bonus
      from sasuser.payrollmaster
      where salary>75000
      order by salary desc;
```

Next, you will add a format for the **Bonus** column. Because the **Bonus** values are dollar amounts, you will use the format *Dollarl2.2*. The *FORMAT=* modifier has been added to the SELECT clause, below, immediately following the column alias **Bonus:**

```
proc sql outobs=15;
   select empid label='Employee ID',
          jobcode label='Job Code',
          salary,
          salary * .10 as Bonus
          format=dollar12.2
   from sasuser.payrollmaster
   where salary>75000
   order by salary desc;
```

Now that column formats and labels have been specified, you can add a title to this PROC SQL query.

specifying Titles and Footnotes

You should already know how to specify and cancel titles and footnotes with other SAS procedures. When you specify titles and footnotes with a PROC SQL query, you must place the TITLE and FOOTNOTE statements in either of the following locations:

- before the PROC SQL statement
- between the PROC SQL statement and the SELECT statement.

In the following PROC SQL query, two title lines have been added between the PROC SQL statement and the SELECT statement:

```
proc sql outobs=15;
title 'Current Bonus Information';
title2 'Employees with Salaries > $75,000';
   select empid label='Employee ID',
          jobcode label='Job Code',
          salary,
          salary * .10 as Bonus
          format=dollar12.2
   from sasuser.payrollmaster
   where salary>75000
   order by salary desc;
```

Now that these changes have been made, you can look at the enhanced query output.

Current Bonus Information Employees with Salaries > $75,000			
Employee ID	Job Code	Salary	Bonus

1118	PT3	$155,931	$15.593.10
1777	PT3	$153,482	$15.348.20
1404	PT2	$127,926	$12.792.60
1107	PT2	$125,968	$12.596.80
1928	PT2	$125,801	$12.580.10
1106	PT2	$125,485	$12.548.50
1333	PT2	$124,048	$12.404.80
1890	PT2	$120,254	$12.025.40
1410	PT2	$118,559	$11.855.90
1442	PT2	$118,350	$11.835.00
1830	PT2	$118,259	$11.825.90
1478	PT2	$117,884	$11,788.40
I556	PT1	$99,889	$9.988.90
1439	PT1	$99,030	$9,903.00
1428	PT1	$96,274	$9,627.40

The first two columns have new labels, the **`Bonus`** values are consistently formatted, and two title lines are displayed at the top of the output.

Adding a Character Constant to Output

Another way of enhancing PROC SQL query output is to define a column that contains a character constant. To do this, you include a text string in quotation marks in the SELECT clause.

> **Tip** You can define a column that contains a numeric constant in a similar way, by listing a numeric value (without quotation marks) in the SELECT clause.

You can look at the preceding PROC SQL query output again and determine where you can add a text string.

Current Bonus Information Employees with Salaries > $75,000			
Employee ID	**Job Code**	**Salary**	**Bonus**
1118	PT3	$155,931	$15.593.10
1777	PT3	$153,482	$15.348.20
1404	PT2	$127,926	$12.792.50
1107	PT2	$125,968	$12.596.80
1928	PT2	$125,801	$12.580.10
1106	PT2	$125,485	$12.548.50
1333	PT2	$124,048	$12.404.80
1890	PT2	$120,254	$12.025.40
1410	PT2	$118,559	$11.855.90
1442	PT2	$118,350	$11.835.00
1830	PT2	$118,259	$11.825.90
1478	PT2	$117,884	$11.788.40
1556	PT1	$99,889	$9.988.90
1439	PT1	$99,030	$9,903.00
1428	PT1	$95,274	$9.627.40

You can remove the column label **`Bonus`** and display the text *bonus is:* in a new column to the left of the **`Bonus`** column. This is how you want the columns and rows to appear in the query output.

Current Bonus Information Employees with Salaries > $75,000				
Employee ID	**Job Code**	**Salary**		
1118	PT3	$155,931	bonus is:	$15.593.10
1777	PT3	$153,482	bonus is:	$15.348.20
1404	PT2	$127,926	bonus is:	$12.792.60
1107	PT2	$125,968	bonus is:	$12.596.80
1928	PT2	$125,801	bonus is:	$12.580.10
1106	PT2	$125,485	bonus is:	$12.548.50
1333	PT2	$124,048	bonus is:	$12.404.80
1890	PT2	$120,254	bonus is:	$12.025.40
1410	PT2	$118,559	bonus is:	$11.855.90
1442	PT2	$118,350	bonus is:	$11.835.00
1830	PT2	$118,259	bonus is:	$11.825.90
1478	PT2	$117,884	bonus is:	$11.788.40
1556	PT1	$99,889	bonus is:	$9.988.90
1439	PT1	$99,030	bonus is:	$9,903.00
1428	PT1	$96,274	bonus is:	$9.627.40

To specify a new column that contains a character constant, you include the text string in quotation marks in the SELECT clause list. Your modified PROC SQL query is shown below:

```
proc sql outobs=15;
title 'Current Bonus Information';
title2 'Employees with Salaries > $75,000';
   select empid label='Employee ID',
          jobcode label='Job Code',
          salary,
          'bonus is:',
          salary * .10 format=dollar12.2
   from sasuser.payrollmaster
   where salary>75000
   order by salary desc;
```

In the SELECT clause list, the text string *bonus is:* has been added between **`Salary`** and **`Bonus`**.

Note that the code **`as Bonus`** has been *removed* from the last line of the SELECT clause. Now that the character constant has been added, the column alias **`Bonus`** is no longer needed.

Summarizing and Grouping Data

Overview

Instead of just listing individual rows, you can use a *summary function* (also called an aggregate function) to produce a statistical summary of data in a table. For example, in the SELECT clause in the following query, the AVG function calculates the average (or mean) miles traveled by frequent-flyer club members. The GROUP BY clause tells PROC SQL to calculate and display the average for each membership group **(`MemberType`)**.

```
proc sql;
   select membertype,
          avg(milestraveled)
          as AvgMilesTraveled
      from sasuser.frequentflyers
      group by membertype;
```

MemberType	AvgMilesTraveled
BRONZE	52938.11

GOLD	48392.82
SILVER	51119.64

You should already be familiar with the list of summary functions that can be used in a PROC SQL query.

PROC SQL calculates summary functions and outputs results in different ways depending on a combination of factors. Four key factors are

- whether the summary function specifies one or multiple columns as arguments
- whether the query contains a GROUP BY clause
- if the summary function is specified in a SELECT clause, whether there are additional columns listed that are outside of a summary function
- whether the WHERE clause, if there is one, contains only columns that are specified in the SELECT clause.

To ensure that your PROC SQL queries produce the intended output, it is important to understand how the factors listed above affect the processing of summary functions. Consider an overview of all the factors, followed by a detailed example that illustrates each factor.

Number of arguments and Summary Function Processing

Summary functions specify one or more arguments in parentheses. In the examples shown in this chapter, the arguments are always columns in the table being queried.

Note The ANSI-standard summary functions, such as AVG and COUNT, can be used only with a single argument. The SAS summary functions, such as MEAN and N, can be used with either single or multiple arguments.

The following chart shows how the number of columns specified as arguments affects the way that PROC SQL calculates a summary function.

If a summary function…	Then the calculation is…	Example
specifies *one column* as argument	performed *down the column*	`proc sql; select avg(salary)as AvgSalary from sasuser.payrollmaster;`
specifies *multiple columns* as arguments	performed *across columns* for each row	`proc sql outobs=10; select sum(boarded,transferred,nonrevenue) as Total from sasuser.marchflights;`

Groups and Summary Function Processing

Summary functions perform calculations on groups of data. When PROC SQL processes a summary function, it looks for a GROUP BY clause:

If a GROUP BY clause…	Then PROC SQL…	Example
is *not present* in the query	applies the function to *the entire table*	`proc sql outobs=10; select jobcode, avg(salary) as AvgSalary from sasuser.payrollmaster;`
is present in the query	applies the function to *each group specified in the GROUPBY clause*	`proc sql outobs=10; select jobcode, avg(salary) as AvgSalary from sasuser.payrollmaster group by jobcode;`

If a query contains a GROUP BY clause, all columns in the SELECT clause that do not contain a summary function should be listed in the GROUP BY clause or unexpected results might be returned.

SELECT Clause Columns and Summary Function Processing

A SELECT clause that contains a summary function can also list additional columns that are not specified in the summary function. The presence of these additional columns in the SELECT clause list causes PROC SQL to display the output differently.

If a SELECT clause…	Then PROC SQL…	Example
contains summary function(s) and *no columns* outside of summary functions	calculates a single value by using the summary function for the entire table or, if groups are specified in the GROUP BY clause, for each group *combines or rolls up the information into a single row of output for the entire table or, if groups are specified, for each group*	`proc sql; select avg(salary) as AvgSalary from sasuser.payrollmaster;`
contains summary function(s) and *additional columns* outside of summary functions	calculates a single value for the entire table or, if groups are specified, for each group, and *displays all rows of output with the single or grouped value(s) repeated*	`proc sql; select jobcode, gender, avg(salary) as AvgSalary from sasuser.payrollmaster group by jobcode,gender;`

Note *WHERE clause columns* also affect summary function processing. If there is a WHERE clause that references only columns that are specified in the SELECT clause, PROC SQL combines information into a single row of output. However, this condition is not covered in this chapter. For more information, see the SAS documentation for the SQL procedure.

In the next few sections, you will look more closely at the query examples shown above to see how the first three factors impact summary function processing.

Compare two PROC SQL queries that contain a summary function: one with a single argument and the other with multiple arguments. To keep things simple, these queries do not contain a GROUP BY clause.

Using a Summary Function with a Single Argument (Column)

Below is a PROC SQL query that displays the average salary of all employees listed in the table *Sasuser. payrollmaster*.

```
proc sql;
   select avg(salary) as AvgSalary
      from sasuser.payrollmaster;
```

The SELECT statement contains the summary function AVG with **`Salary`** as its argument. Because there is only *one* column as an argument, the function calculates the statistic *down the Salary column* to display a single value: the average salary for all employees. The output is shown here.

AvgSalary
54079.62

Using a Summary Function with Multiple Arguments (Columns)

Consider a PROC SQL query that contains a summary function with *multiple* columns as arguments. This query calculates the total number of Passengers for each flight in March by adding the number of boarded, transferred, and nonrevenue passengers:

```
proc sql outobs=10;
   select sum(boarded,transferred,nonrevenue)
          as Total
      from sasuser.marchflights;
```

The SELECT clause contains the summary function SUM with three columns as arguments. Because the function contains multiple arguments, the statistic is calculated *across the three columnsfor each row* to produce the following output.

Total
123
196
167
222
185
163
119
200
70
227

Note Without the OUTOBS= option, *all* rows in the table would be displayed in the output.

Consider how a PROC SQL query with a summary function is affected by including a GROUP BY clause and including columns outside of a summary function.

Using a Summary Function without a GROUP BY Clause

Once again, here is the PROC SQL query that displays the average salary of all employees listed in the table *Sasuser.payrollmaster*. This query contains a summary function but, since the goal is to display the average across *all* employees, there is no GROUP BY clause.

```
proc sql outobs=20;
   select avg(salary) as AvgSalary
    from sasuser.payrollmaster;
```

Note that the SELECT clause lists only one column: a new column that is defined by a summary function calculation. There are no columns listed outside of the summary function.

Here is the query output.

AvgSalary
54079.62

Using a Summary Function with Columns Outside of the Function

Suppose you calculate an average for each job group and group the results by job code. Your first step is to add an existing column **(**`JobCode`**)** to the SELECT clause list. The modified query is shown here:

```
proc sql outobs=20;
   select jobcode, avg(salary) as AvgSalary
    from sasuser.payrollmaster;
```

Consider what the query output looks like now that the SELECT statement contains a column **(**`JobCode`**)** that is *not* a summary function argument.

JobCode	AvgSalary
TA2	54079.62
ME2	54079.62
ME1	54079.62
FA3	54079.62
TA3	54079.62
ME3	54079.62

SCP	54079.62
PT2	54079.62
TA2	54079.62
TA3	54079.62
ME1	54079.62
PT1	54079.62
SCP	54079.62
PT1	54079.62
ME2	54079.62
ME2	54079.62
BCK	54079.62
FA2	54079.62
SCP	54079.62
PT2	54079.62

Note Remember that this PROC SQL query uses the OUTOBS= option to limit the output to 20 rows. Without this limitation, the output of this query would display all 148 rows in the table.

As this result shows, adding a column to the SELECT clause that is *not* within a summary function causes PROC SQL to output *all* rows instead of a single value. To generate this output, PROC SQL

- calculated the average salary down the column as a single value *(54079.62)*
- displayed *all* rows in the output, because **`JobCode`** is not specified in a summary function.

Therefore, the single value for **`AvgSalary`** is *repeated* for each row.

Note When this query is submitted, the SAS log displays a message indicating that data remerging has occurred. Data remerging is explained later in this chapter.

While this result is interesting, you have not yet reached your goal: grouping the data by **`JobCode`**. The next step is to add the GROUP BY clause.

Using a Summary Function with a GROUP BY Clause

Below is the PROC SQL query from the previous page, to which has been added a GROUP BY clause that specifies the column **`JobCode`**. (In the SELECT clause, **`JobCode`** is specified but is *not* used as a summary function argument.) Other changes to the query include removing the OUTOBS= option (it is unnecessary) and specifying a format for the **`AvgSalary`** column.

```
proc sql;
   select jobcode,
          avg(salary) as AvgSalary format=dollar11.2
      from sasuser.payrollmaster
      group by jobcode;
```

Consider how the addition of the GROUP BY clause affects the output.

JobCode	AvgSalary
BCK	$36,111.89
FA1	$32,255.18
FA2	$33,181.50
FA3	$46,107.57
ME1	$33,900.50
ME2	$49,807.64

ME3	$53,374.86
NA1	$58,845.00
NA2	$73,336.00
PT1	$95,071.13
PT2	$122,253.40
PT3	$154,706.50
SCP	$25,632.14
TA1	$38,803.89
TA2	$47,004.90
TA3	$55,551.42

Success! The summary function has been calculated for each **`JobCode`** group, and the results are grouped by **`JobCode`**.

Counting Values by Using the COUNT Summary Function

Sometimes you want to count the number of rows in an entire table or in groups of rows. In PROC SQL, you can use the COUNT summary function to count the number of rows that have nonmissing values. There are three main ways to use the COUNT function.

Using this form of COUNT…	Returns…	Example
COUNT(*)	the *total number of rows* in a group or in a table	`select count(*) as Count`
COUHI(*column*)	the *total number of rows in a group or in a table for which there is a nonmissing value in the selected column*	`select count(jobcode) as Count`
COUNT(DISTINCT *column*)	the *total number of unique values* in a column	`select count(distinct jobcode) as Count`

Caution The COUNT summary function counts only the *nonmissing* values; missing values are ignored. Many other summary functions also ignore missing values. For example, the AVG function returns the average of the nonmissing values only. When you use a summary function with data that contains missing values, the results might not provide the information that you expect. It is a good idea to familiarize yourself with the data before you use summary functions in queries.

Tip To count the number of missing values, use the NMISS function. For more information about the NMISS function, see the SAS documentation.

Consider the three ways of using the COUNT function.

Counting All Rows

Suppose you want to know how many employees are listed in the table *Sasuser. Payrollmaster*. This table contains a separate row for each employee, so counting the number of rows in the table gives you the number of employees. The following PROC SQL query accomplishes this task:

```
proc sql;
   select count(*) as Count
    from sasuser.payrollmaster;
```

Count
148

Note The COUNT summary function is the only function that allows you to use an asterisk (*) as an argument.

You can also use COUNT(*) to count rows within groups of data. To do this, you specify the groups in the GROUP BY clause. Consider a more complex PROC SQL query that uses COUNT(*) with grouping. This time, the goal is to find the total number of employees within each job category, using the same table that is used above.

```
proc sql;
   select substr(jobcode,1,2)
          label='Job Category',
          count(*) as Count
      from sasuser.payrollmaster
      group by 1;
```

This query defines two new columns in the SELECT clause. The first column, which is labeled **JobCategory**, is created by using the SAS function SUBSTR to extract the two-characterjob category from the existing **JobCode** field. The second column, **Count**, is created by using the COUNT function. The GROUP BY clause specifies that the results are to be grouped by the first defined column (referenced by 1 because the column was not assigned a name).

Job Category	Count
BC	9
FA	34
ME	29
NA	8
PT	20
SC	7
TA	41

Caution When a column contains missing values, PROC SQL treats the missing values as a single group. This can sometimes produce unexpected results.

Counting All Non-Missing Values in a Column

Suppose you want to count all of the non-missing values in a specific column instead of in the entire table. To do this, you specify the name of the column as an argument of the COUNT function. For example, the following PROC SQL query counts all non-missing values in the column **JobCode:**

```
proc sql;
   select count(JobCode) as Count
      from sasuser.payrollmaster;
```

Count
148

Because the table has no missing data, you get the same output with this query as you would by using COUNT(*). **JobCode** has a non-missing value for each row in the table. However, if the **JobCode** column contained missing values, this query would produce a lower value of **Count** than the previous query. For example, if **JobCode** contained three missing values, the value of **Count** would be 145.

Counting All Unique Values in a Column

To count all unique values in a column, add the keyword *DISTINCT* before the name of the column that is used as an argument. For example, here is the previous query modified to count only the unique values:

```
proc sql;
   select count(distinct jobcode) as Count
      from sasuser.payrollmaster;
```

This query counts 16 unique values for **JobCode**.

Count
16

To display the unique **JobCode** values, you can apply the method of eliminating duplicates, which was discussed earlier.

The following query lists only the unique values for **JobCode**.

```
proc sql;
   select distinct jobcode
    from sasuser.payrollmaster;
```

There are 16 job codes, so the output contains 16 rows.

JobCode
BCK
FA1
FA2
FA3
ME1
ME2
ME3
NA1
NA2
PT1
PT2
PT3
SCP
TA1
TA2
TA3

Selecting Groups by Using the HAVING Clause

You have seen how to use the GROUP BY clause to group data. For example, the following query calculates the average salary within each job-code group, and displays the average for each job code:

```
proc sql;
   select jobcode,
          avg(salary) as AvgSalary
          format=dollar11.2
      from sasuser.payrollmaster
      group by jobcode;
```

There are 16job codes in the table, so the output displays 16 rows.

JohCode	AvgSalary
BCK	$36,111.89
FA1	$32,255.18
FA2	$39,181.50
FA3	$46,107.57
ME1	$39,900.50
ME2	$49,807.64
ME3	$59,374.86
NA1	$58,845.00
NA2	$73,336.00
PT1	$95,071.13
PT2	$122,253.40
PT3	$154,706.50

$CP	$25,632.14
TA1	$38,809.89
TA2	$47,004.90
TA3	$55,551.42

Now, suppose you want to select only a *subset of groups* for your query output. You can use a *HAVING* clause, following a GROUP BY clause, to select (or filter) the groups to be displayed. The way a HAVING clause affects groups is similar to the way that a WHERE clause affects individual rows. As in a WHERE clause, the HAVING clause contains an expression that is used to subset the data. Any valid SAS expression can be used. When you use a HAVING clause, PROC SQL displays only the groups that satisfy the HAVING expression.

Note You can use summary functions in a HAVING clause but not in a WHERE clause, because a HAVING clause is used with groups, but a WHERE clause can be used only with individual rows.

Modify the query shown above so that it selects only the **JobCode** groups with an average salary of more than $56,000. The HAVING clause has been added at the end of the query.

```
proc sql;
   select jobcode,
          avg(salary) as AvgSalary
          format=dollar11.2
      from sasuser.payrollmaster
      group by jobcode
      having avg(salary) > 56000;
```

Tip Alternatively, because the average salary is already calculated in the SELECT clause, the HAVING clause could specify the column alias **AvgSalary** :

```
having AvgSalary > 560 00
```

Note that you do *not* have to specify the keyword CALCULATED in a HAVING clause; you would have to specify it in a WHERE clause.

The query output is shown below. This output is smaller than the previous output, because only a subset of the job-code groups is displayed.

JohCode	AvgSalary
ME3	$59,374.86
NA1	$58,845.00
NA2	$73,336.00
PT1	$95,071.13
PT2	$122,253.40
PT3	$154,706.50

If you omit the GROUP BY clause in a query that contains a HAVING clause, then the HAVING clause and summary functions (if any are specified) treat the entire table as one group. Without a GROUP BY clause, the HAVING clause in the example shown above calculates the average salary for the table as a whole (all jobs in the company), not for each group (each job code). The output contains either all the rows in the table (if the average salary for the entire table is greater than $56,000) or none of the rows in the table (if the average salary for the entire table is less than $56,000).

Understanding Data Remerging

Sometimes, when you use a summary function in a SELECT clause or a HAVING clause, PROC SQL must *remerge* data (make two passes through the table). Remerging requires additional processing time and is of ten unavoidable. However, there are some situations in which you might be able to modify your query to avoid remerging. Understanding how and when remerging occurs will increase your ability to write efficient queries.

Consider a PROC SQL query that requires remerging. This query calculates each navigator's salary as a percentage of all navigators' salaries:

```
proc sql;
   select empid, salary,
          (salary/sum(salary)) as Percent
          format=percent8.2
      from sasuser.payrollmaster
      where jobcode contains 'NA';
```

When you submit this query, the SAS log displays the following message.

Table 2.4: SAS Log

```
NOTE: The query requires remerging summary statistics back with the original data.
```

Remerging occurs whenever any of the following conditions exist:

- The values returned by a summary function are used in a calculation.
- The SELECT clause specifies a column that contains a summary function *and other column(s)* that are *not* listed in a GROUP BY clause.
- The HAVING clause specifies one or more columns or column expressions that are *not* included in a subquery or a GROUP BY clause.

During remerging, PROC SQL makes two passes through the table:

1. PROC SQL calculates and returns the value of summary functions. PROC SQL also groups data according to the GROUP BY clause.
2. PROC SQL retrieves any additional columns and rows that it needs to display in the output, and uses the result from the summary function to calculate any arithmetic expressions in which the summary function participates.

Example

Consider how PROC SQL remerges data when it processes the following query:

```
proc sql;
   select empid, salary,
          (salary/sum(salary)) as Percent
          format=percent8.2
      from sasuser.payrollmaster
      where jobcode contains 'NA';
```

In the *first pass*, for each row in which the jobcode contains **'NA '**, PROC SQL calculates and returns the value of the SUM function (specified in the SELECT clause).

In the *second pass*, PROC SQL retrieves the additional columns and rows that it needs to display in output **(EmpID, Salary)** and the rows in which **JobCode** contains 'NA'. PROC SQL also uses the result from the SUM function to calculate the arithmetic expression **(salary/sum(salary))**.

Caution Some implementations of SQL do not support remerging and would consider the preceding example to be in error.

Tip You can obtain the same results by using a subquery. Subqueries are discussed later in this chapter.

Subsetting Data by Using Subqueries

Introducing Subqueries

The WHERE and HAVING clauses both subset data based on an expression. In the query examples shown earlier in this chapter, the WHERE and HAVING clauses contained standard SAS expressions. For example, the expression in the following WHERE clause uses the BETWEEN-AND conditional operator and specifies the **Salary** column as an operand:

```
where salary between 70000 and 80000
```

PROC SQL also offers another type of expression that can be used for subsetting in WHERE and HAVING clauses: a query-expression or *subquery*. A subquery is a query that is nested in, and is part of, another query. A PROC SQL query might contain subqueries at one or more levels.

Note Subqueries are also known as nested queries, inner queries, and sub-selects.

The following PROC SQL query contains a subquery in the HAVING clause that returns all job codes where the average salary for that job code is greater than the company average salary.

```
proc sql;
   select jobcode,
          avg(salary) as AvgSalary
          format=dollar11.2
      from sasuser.payrollmaster
      group by jobcode
      having avg(salary) >
         (select avg(salary)
            from sasuser.payrollmaster);
```

Tip It is recommended that you enclose a subquery (inner query) in parentheses, as shown here.

A subquery selects one or more rows from a table, and then returns single or multiple values to be used by the outer query. The subquery shown above is a *single-value subquery;* it returns a single value, the average salary from the table *Sasuser.payrollmaster*, to the outer query. A subquery can return values for *multiple rows* but only for a *single column*.

The table that a subquery references can be either the same as or different from the table referenced by the outer query. In the PROC SQL query shown above, the subquery selects data from the same table as the outer query.

Types of Subqueries

There are two types of subqueries.

Type of Subquery	Description
noncorrelated	a self-contained subquery that *executes independently of the outer query*
correlated	a dependent subquery that *requires one or more values to be passed to it by the outer query* before the subquery can return a value to the outer query

Both noncorrelated and correlated subqueries can return either single or multiple values to the outer query.

The next few sections provide a more in-depth look at noncorrelated and correlated subqueries, and how they are processed.

Subsetting Data by Using Noncorrelated Subqueries

A noncorrelated subquery is a self-contained subquery that executes independently of the outer query.

Using Single-Value Noncorrelated Subqueries

The simplest type of subquery is a *noncorrelated subquery that returns a single value*.

The following PROC SQL query is the same query that is used in the previous section. This query dis plays job codes for which the group's average salary exceeds the company's average salary. The HAVING clause contains a noncorrelated subquery.

```
proc sql;
   select jobcode,
          avg(salary) as AvgSalary
          format=dollar11.2
      from sasuser.payrollmaster
      group by jobcode
      having avg(salary) >
         (select avg(salary)
            from sasuser.payrollmaster);
```

PROC SQL always evaluates a noncorrelated subquery before the outer query. If a query contains noncorrelated subqueries at more than one level, PROC SQL evaluates the innermost subquery first and works outward, evaluating the outermost query last.

In the query shown above, the inner query and outer query are processed as follows:

1. To complete the expression in the HAVING clause, the subquery calculates the average salary for the entire company (all rows in the table), using the AVG summary function with **`Salary`** as an argument.
2. The subquery returns the value of the average salary to the outer query.
3. The outer query calculates the average salary (in the SELECT clause) for each **`JobCode`** group (as defined in the GROUP BY clause), and selects only the groups whose average salary is greater than the company's average salary.

The query output is shown here.

JobCode	Avg Salary
ME3	$59,374.86
NA1	$58,845.00
NA2	$73,336.00
PT1	$95,071.13
PT2	$122,253.40
PT3	$154,706.50
TA3	$55,551.42

This noncorrelated subquery returns only a single value, the average salary for the whole company, to the outer query. Both the subquery and the outer query use the same table as a source.

Using Multiple-Value Noncorrelated Subqueries

Some subqueries are *multiple-value subqueries;* they return more than one value (row) to the outer query. If your noncorrelated subquery might return a value for more than one row, be sure to use one of the following operators in the WHERE or HAVING clause that can handle multiple values:

- the conditional operator IN
- a comparison operator that is modified by ANY or ALL
- the conditional operator EXISTS.

Caution If you create a noncorrelated subquery that returns multiple values, but the WHERE or HAVING clause in the outer query contains an operator other than one of the operators that are specified above, the query will fail. An error message is displayed in the SAS log, which indicates that the subquery evaluated to more than one row. For example, if you use the equal (=) operator with a noncorrelated subquery that returns multiple values, the query will fail. The equal operator can handle only a single value.

Consider a query that contains both the conditional operator IN and a noncorrelated subquery that returns multiple values. (The operators ANY, ALL, and EXISTS are presented later in this chapter.)

Example

Suppose you want to send birthday cards to employees who have birthdays coming up. You decide to create a PROC SQL query that will list the names and addresses of all employees who have birthdays in February. This query, unlike the one shown on the previous page, will select data from two different tables:

- employee names and addresses in the table *Sasuser.Staffmaster*
- employee birth dates in the table *Sasuser.payrollmaster*.

In both tables, the employees are identified by their employee identification number **(`EmpID`)**.

In the following PROC SQL query, the WHERE clause contains the conditional operator IN followed by a noncorrelated subquery:

```
proc sql;
   select empid, lastname, firstname,
          city, state
      from sasuser.staffmaster
      where empid in
         (select empid
            from sasuser.payrollmaster
            where month(dateofbirth)=2);
```

This query is processed as follows.

- To complete the expression in the WHERE clause of the outer query, the subquery selects the employees whose date of birth is February. Note that the MONTH function is used in the subquery.
- The subquery then returns the **`EmpID`** values of the selected employees to the outer query.
- The outer query displays data (from the columns identified in the SELECT clause) for the employees identified by the subquery.

The output, shown below, lists the six employees who have February birthdays.

EmpID	LastName	FirstName	City	State
1403	BOWDEN	EARL	BRIDGEPORT	CT
1404	CARTER	DONALD	NEW YORK	NY
1834	LONG	RUSSELL	NEW YORK	NY
1103	MCDANIEL	RONDA	NEW YORK	NY
1420	ROUSE	JEREMY	PATERSON	NJ
1390	SMART	JONATHAN	NEW YORK	NY

Although an innerjoin would have generated the same results, it is better to use a subquery in this example since no columns from the **`sasuser.payrollmater`** table were in the output.

Using Comparisons with Subqueries

Sometimes it is helpful to compare a value with a set of values returned by a subquery. When a subquery might return multiple values, you must use one of the conditional operators ANY or ALL to modify a comparison operator in the WHERE or HAVING clause immediately before the subquery. For example, the following WHERE clause contains the less than (<) comparison operator and the conditional operator ANY:

```
where dateofbirth < any
   <subquery...>
```

Caution If you create a noncorrelated subquery that returns multiple values, and if the WHERE or HAVING clause in the outer query contains a comparison operator that is not modified by ANY or ALL, the query will fail.

When the outer query contains a comparison operator that is modified by ANY or ALL, the outer query compares each value that it retrieves against the value(s) returned by the subquery. All values for which the comparison is true are then included in the query output. If ANY is specified, then the comparison is true if it is true for any one of the values that are returned by the subquery. If ALL is specified, then the comparison is true only if it is true for all values that are returned by the subquery.

Note The operators ANY and ALL can be used with correlated subqueries, but they are usually used only with noncorrelated subqueries.

Consider how the operators ANY or ALL are used.

Using the ANY Operator

An outer query that specifies the ANY operator selects values that pass the comparison test with *any* of the values that are returned by the subquery.

For example, suppose you have an outer query containing the following WHERE clause:

```
where dateofbirth < any
   <subquery...>
```

This WHERE clause specifies that **`DateofBirth`** (the operand) should be less than *any* (the comparison operator) of the values returned by the subquery.

The following chart shows the effect of using ANY with these common comparison operators: greater than (>), less than (<) and equal to (=).

Comparison Operator with ANY	Outer Query Selects…	Example
> ANY	values that are *greater than any value* returned by the subquery	If the subquery returns the values 20, 30, 40, then the outer query selects all values that are > 20 (the lowest value that was returned by the subquery).
<ANY	values that are *less than any value* returned by the subquery	If the subquery returns the values 20, 30, 40, then the outer query selects all values that are < 40 (the highest value that was returned by the subquery).
= ANY	values that are *equal to any value* returned by the subquery	If the subquery returns the values 20, 30, 40, the outer query selects all values that are = 20 or = 30 or = 40.

Tip Instead of using the ANY operator with a subquery, there are some SAS functions that you can use to achieve the same result with greater efficiency. Instead of> ANY, use the MIN function in the subquery. Instead of<ANY, use the MAX function in the subquery.

Example

Suppose you want to identify *any* flight attendants at level 1 or level 2 who are older than *any* of the flight attendants at level 3. Job type and level are identified in **`JobCode;`** each flight attendant has the job code *FAI, FA2*, or *FA3*. The following PROC SQL query accomplishes this task by using a subquery and the ANY operator:

```
proc sql;
   select empid, jobcode, dateofbirth
    from sasuser.payrollmaster
    where jobcode in ('FA1','FA2')
          and dateofbirth < any
             (select dateofbirth
                from sasuser.payrollmaster
                where jobcode='FA3');
```

Here is what happens when this query is processed:

- The subquery returns the birthdates of all level-3 flight attendants.
- The outer query selects only those level-1 and level-2 flight attendants whose birthdate is *less than any* of the dates returned by the subquery.

Note that both the outer query and subquery use the same table.

Note Internally, SAS represents a date value as the number of days from January 1, I960, to the given date. For example, the SAS date for 17 October 1991 is 11612. Representing dates as the number of days from a reference date makes it easy for the computer to store them and perform calendar calculations. These numbers are not meaningful to users, however, so several formats are available for displaying dates and datetime values in most of the commonly used notations.

Below are the query results.

EiripID	JobCode	DateOfBirth
1574	FA2	01MAY1958
1475	FA2	19DEC1959

1124	FA1	13JUL1956
1422	FA1	08JUN1962
1368	FA2	15JUN1959
1411	FA2	31MAY1959
1477	FA2	25MAR1962
1970	FA1	29SEP1962
1413	FA2	20SEP1963
1434	FA2	14JUL1960
1390	FA2	23FEB1963
1135	FA2	24SEP1958
1415	FA2	12MAR1956
1122	FA2	04MAY1961

Tip Using the ANY operator to solve this problem results in a large number of calculations, which increases processing time. For this example, it would be more efficient to use the MAX function in the subquery. The alternative WHERE clause follows:

```
where jobcode in ('FA1','FA2')
      and dateofbirth <
         (select max(dateofbirth)
            from […]
```

For more information about the MAX function, see the SAS documentation.

Using the ALL Operator

An outer query that specifies the ALL operator selects values that pass the comparison test with *all* of the values that are returned by the subquery.

The following chart shows the effect of using ALL with these common comparison operators: greater than (>) and less than (<).

Comparison Operator with ALL	Sample Values Returned by Subquery	Signifies…
> ALL	(20, 30, 40)	> 40 (greater than the highest number in the list)
< ALL	(20, 30, 40)	< 20 (less than the lowest number in the list)

Example

Substitute ALL for ANY in the previous query example. The following query identifies level-1 and level-2 flight attendants who are older than *all* of the level-3 flight attendants:

```
proc sql;
   select empid, jobcode, dateofbirth
      from sasuser.payrollmaster
      where jobcode in ('FA1','FA2')
            and dateofbirth < all
               (select dateofbirth
                  from sasuser.payrollmaster
                  where jobcode='FA3');
```

Here is what happens when this query is processed:

1. The subquery returns the birthdates of all level-3 flight attendants.
2. The outer query selects only those level-1 and level-2 flight attendants whose birthdate is less than *all* of the dates returned by the subquery.

The query results, below, show that only two level-1 or level-2 flight attendants are older than all of the level-3 flight attendants.

EmpID	JobCode	DateOfBirth
1124	FA1	13JUL1956
1415	FA2	12MAR1956

Tip For this example, it would be more efficient to solve this problem using the MIN function in the subquery instead of the ALL operator. The alternative WHERE clause follows:

```
where jobcode in ('FA1','FA2')
      and dateofbirth <
        (select min(dateofbirth)
            from […]
```

For more information about the MIN function, see the SAS documentation.

Subsetting Data by Using Correlated Subqueries

Overview

Correlated subqueries *cannot be evaluated independently*, but depend on the values passed to them by the outer query for their results. Correlated subqueries are evaluated for *each* row in the outer query and, therefore, tend to require more processing time than noncorrelated subqueries.

Note Usually, a PROC SQL join is a more efficient alternative to a correlated subquery. You should already be familiar with basic PROC SQL joins.

Example

Consider an example of a PROC SQL query that contains a correlated subquery. The following query displays the names of all navigators who are also managers. The WHERE clause in the subquery lists the column **`Staffmaster.EmpID`**, which is the column that the outer query must pass to the correlated subquery.

```
proc sql;
   select lastname, firstname
      from sasuser.staffmaster
      where 'NA'=
            (select jobcategory
               from sasuser.supervisors
               where staffmaster.empid =
                     supervisors.empid);
```

Note When a column appears in more than one table, the column name is preceded by the table name or alias to avoid ambiguity. In this example, **`EmpID`** appears in both tables, so the appropriate table name is specified in front of each reference to that column.

The output from this query is shown below. There are three navigators who are also managers.

LastName	FirstName
FERNANDEZ	KATRINA
NEWKIRK	WILLIAM
RIVERS	SIMON

Using the EXISTS and NOT EXISTS Conditional Operators

In the WHERE clause or in the HAVING clause of an outer query, you can use the EXISTS or NOT EXISTS conditional operator to test for the existence or non-existence of a set of values returned by a subquery.

Condition	Is true if…
EXISTS	the subquery returns *at least one row*

NOT EXISTS	the subquery returns *no data*

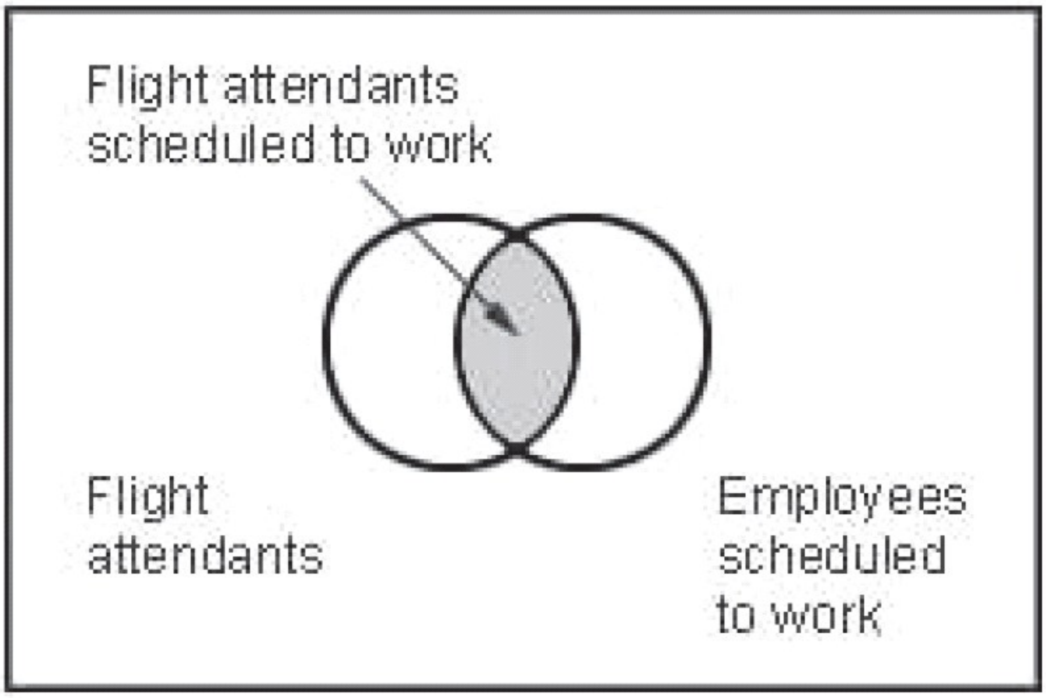
Note The operators EXISTS and NOT EXISTS can be used with both correlated and noncorrelated subqueries.

Example: Correlated Subquery with NOT EXISTS

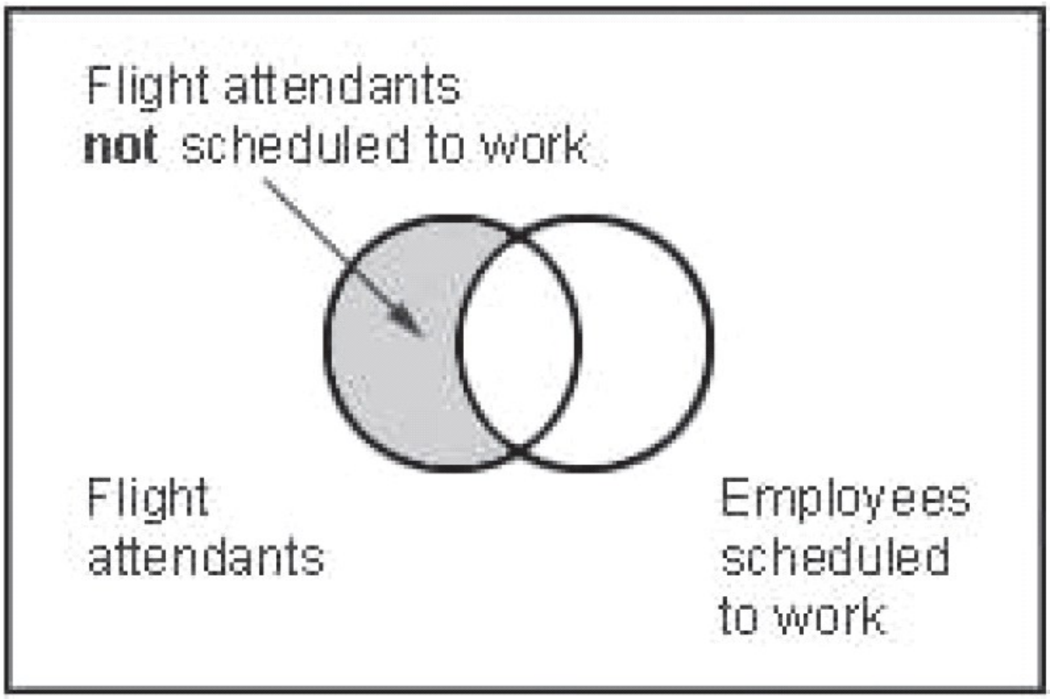
Consider a sample PROC SQL query that includes the NOT EXISTS conditional operator. Suppose you are working with the following tables:

- *Sasuser.Flightattendants* contains the names and employee ID numbers of all flight attendants.
- *Sasuser.Flightschedule* contains one row for each crew member assigned to a flight for each date.

As shown in the diagram below, the intersection of these two tables contains data for all flight attendants who have been scheduled to work.


Flight attendants scheduled to work
Flight attendants
Employees scheduled to work

Now suppose you want to list by name the flight attendants who have not been scheduled. That is, you want to identify the data in the area highlighted below



The following PROC SQL query accomplishes this task by using a correlated subquery and the NOT EXISTS operator

```
proc sql;
   select lastname, firstname
   from sasuser.flightattendants
   where not exists
     (select *
        from sasuser.flightschedule
                    where flightattendants.empid=
                          flightschedule.empid);
```

The output is shown below.

LastName	FirstName
PATTERSON	RENEE
VEGA	FRANKLIN

Validating Query Syntax

Overview

When you are building a PROC SQL query, you might find it more efficient to check your query without actually executing it. To verify the syntax and the existence of columns and tables that are referenced in the query without executing the query, use either of the following:

- the *NOEXEC option* in the PROC SQL statement
- the *VALIDATE keyword* before a SELECT statement.

Consider how you specify the NOEXEC option and the VALIDATE keyword, and examine the minor differences between them.

Using the NOEXEC Option

The NOEXEC option is specified in the following PROC SQL statement:

```
proc sql noexec;
   select empid, jobcode, salary
      from sasuser.payrollmaster
      where jobcode contains 'NA'
      order by salary;
```

If the query is valid and all referenced columns and tables exist, the SAS log displays the following message.

Table 2.5: SAS Log

```
NOTE: Statement not executed due to NOEXEC option.
```

Or, if there are any errors in the query, SAS displays the standard error messages in the log.

When you invoke the NOEXEC option, SAS cheeks the syntax of *all* queries in that PROC SQL step for accuracy but does not execute them.

Using the VALIDATE Keyword

You specify the VALIDATE key word just before a SELECT statement; it is not used with any other PROC SQL statement.

We will modify the preceding PROC SQL query by using the VALIDATE keyword instead of the NOEXEC option:

```
proc sql;
   validate
   select empid, jobcode, salary
      from sasuser.payrollmaster
      where jobcode contains 'NA'
      order by salary;
```

Note Note that the VALIDATE keyword is *not* followed by a semicolon. If the query is valid, the SAS log displays the following message.

Table 2.6: SAS Log

```
NOTE: PROC SQL statement has valid syntax.
```

If there are errors in the query, SAS displays the standard error messages in the log.

The main difference between the VALIDATE keyword and the NOEXEC option is that the VALIDATE keyword only affects the SELECT statement that immediately follows it, whereas the NOEXEC option applies to *all* queries in the PROC SQL step. If You are working with a PROC SQL query that contains multiple SELECT statements, the VALIDATE keyword must be specified before *each* SELECT statement that you want to check.

Additional Features

In addition to the SELECT statement, PROC SQL supports the following statements.

Statement	Use to …
ALTER TABLE *expression*;	add, drop, and modify columns in a table
CREATE *expression*;	build new tables, views, or indexes
DELETE *expression*;	eliminate unwanted rows from a table or view
DESCRIBE *expression*;	display table and view attributes
DROP *expression*;	eliminate entire tables, views, or indexes
INSERT *expression*	add rows of data to tables or views
RESET *<option(s)>;*	add to or change PROC SQL options without re-invoking the procedure
UPDATE *expression*;	modify data values in existing rows of a table or view

You can learn more about these PROC SQL statements in the following chapters:

- "Combining Tables Horizontally Using PROC SQL" on page 86
- "Combining Tables Vertically Using PROC SQL" on page 132

- "Creating and Managing Tables Using PROC SQL" on page 175
- "Creating and Managing Indexes Using PROC SQL" on page 238
- "Creating and Managing Views Using PROC SQL" on page 260
- Managing Processing Using PROC SQL" on page 278

Summary

Contents

This section contains the following topics.

Text Summary

Viewing SELECT Statement Syntax

The SELECT statement and its subordinate clauses are the building blocks that you use to construct all PROC SQL queries.

Displaying All Columns

To display all columns in the order in which they are stored in the table, use an asterisk (*) in the SELECT clause. To write the expanded list of columns to the SAS log, use the FEEDBACK option in the PROC SQL statement.

Limiting the Number of Rows Displayed

To limit the number of rows that PROC SQL displays as output, use the OUTOBS=<< option in the PROC SQL statement.

Eliminating Duplicate Rows from Output

To eliminate duplicate rows from your query results, use the keyword DISTINCT in the SELECT clause.

Subsetting Rows by Using Conditional Operators

In a PROC SQL query, use the WHERE clause with any valid SAS expression to subset data. The SAS expression can contain one or more operators, including the following conditional operators:

- the BETWEEN-AND operator selects within an inclusive range of values
- the CONTAINS or ? operator selects a character string
- the IN operator selects from a list of fixed values
- the IS MISSING or IS NULL operator selects missing values
- the LIKE operator selects a pattern
- the sounds-like (=*) operator selects a spelling variation

Subsetting Rows by Using Calculated Values

It is important to understand how PROC SQL processes calculated columns. When you use a column alias in the WHERE clause to refer to a calculated value, you must use the keyword CALCULATED with the alias.

Enhancing Query Output

You can enhance PROC SQL query output by using SAS enhancements such as column formats and labels, titles and footnotes, and character constraints.

Summarizing and Grouping Data

PROC SQL calculates summary functions and outputs results differently, depending on a combination of factors:

- whether the summary function specifies one or more multiple columns as arguments
- whether the query contains a GROUP BY clause
- if the summary function is specified in a SELECT clause, whether there are additional columns listed that are outside the summary function
- whether the WHERE clause, if there is one, contains only columns that are specified in the SELECT clause.

To count non-missing values, use the COUNT summary function.

To select the groups to be displayed, use a HAVING clause following a GROUP BY clause.

When you use a summary function in a SELECT clause or a HAVING clause, in some situations, PROC SQL must remerge data. When PROC SQL remerges data, it makes two passes through the data, and this requires additional processing time.

Subsetting Data by Using Subqueries

In the WHERE or the HAVING clause of a PROC SQL query, you can use a subquery to subset data. A subquery is a query that is nested in, and is part of, another query. Subqueries can return values from a single row or multiple rows to the outer query but can return values only from a single column.

Subsetting Data by Using Noncorrelated Subqueries

Noncorrelated subqueries execute independently of the outer query. You can use noncorrelated subqueries that return a single value or multiple values. To further qualify a comparison specified in a WHERE or a HAVING clause, you can use the conditional operators ANY and ALL immediately before a noncorrelated (or correlated) subquery.

Subsetting Data by Using Correlated Subqueries

Correlated subqueries cannot be evaluated independently because their results are dependent on the values returned by the outer query. In the WHERE or the HAVING clause of an outer query, you can use the EXISTS and NOT EXISTS conditional operators to test for the existence or non-existence of a set of values returned by the subquery.

Validating Query Syntax

To check the validity of the query syntax without actually executing the query, use the NOEXEC option or the VALIDATE keyword.

Additional Features

PROC SQL supports many statements in addition to the SELECT statement.

Syntax

```
PROC SQL OUTOBS=n;
     SELECT column-l<,…column-n>
          FROM table-1 | view-l<, … table-n | view-n>
          <WHERE expression>
          <GROUP BYcolumn-l<, … column-n>>
          <HAVINGexpression>
          <ORDER BYcolumn-l<, …column-n>>;
QUIT;
sql-expression <NOT> BETWEEN sql-expression AND sql-expression
sql-expression <NOT> CONTAINS sql-expression
sql-expression <NOT> IN (query-expression | constant-l<,…constant-n>)
sql-expression IS MISSING
sql-expression IS NULL
sql-expression <NOT> LIKE sql-expression
sql-expression =* sql-expression
```

Sample Programs

Displaying All Columns in Output and an Expanded Column List in the SAS Log

```
proc sql feedback;
   select *
      from sasuser.staffchanges;
quit;
```

Eliminating Duplicate Rows from Output

```
proc sql;
   select distinct flightnumber, destination
      from sasuser.internationalflights
      order by 1;
quit;
```

Subsetting Rows by Using Calculated Values

```
proc sql outobs=10;
   validate
   select flightnumber,
          date label="Flight Date", destination,
          boarded + transferred + nonrevenue
          as Total
      from sasuser.marchflights
      where calculated total between 100 and 150;
quit;
```

Subsetting Data by Using a Noncorrelated Subquery

```
proc sql noexec;
   select jobcode,
          avg(salary) as AvgSalary
      format=dollar11.2 from sasuser.payrollmaster
      group by jobcode
      having avg(salary) >
         (select avg(salary)
            from sasuser.payrollmaster);
quit;
```

Subsetting Data by Using a Correlated Subquery

```
proc sql;
title 'Frequent Flyers Who Are Not Employees';
   select count(*) as Count
      from sasuser.frequentflyers
      where not exists
          (select *
             from sasuser.staffmaster
             where name=
                   trim(lastname)||', '||firstname);
quit;
```

Points to Remember

- When you use summary functions, look for missing values. If a table contains missing values, your results might not be what you expect. Many summary functions ignore missing values when performing calculations, and PROC SQL treats missing values in a column as a single group.
- When you create complex queries, it is helpful to use the NOEXEC option or the VALIDATE statement to validate your query without executing it.

Quiz

Select the best answer for each question. After completing the quiz, check your answers using the answer key in the appendix.

1. Which PROC SQL query will remove duplicate values of **`MemberType`** from the query output, so that only the unique values are listed? ?

```
a. proc sql nodup;
   select membertype
      from sasuser.frequentflyers;

b. proc sql;
   select distinct(membertype)
          as MemberType
      from sasuser.frequentflyers;

c. proc sql;
   select unique membertype
      from sasuser.frequentflyers
      group by membertype;

d. proc sql;
    select distinct membertype
       from sasuser.frequentflyers;
```

2. Which of the following will cause PROC SQL to list rows that have *no* data in the **`Address`** column? ?

a. `WHERE address is missing`

b. `WHERE address not exists`

c. `WHERE address is null`

d. both a and c

3. You are creating a PROC SQL query that will list all employees who have spent (or overspent) their allotted 120 hours of vacation for the current year. The hours that each employee used are stored in the existing column **`Spent.`** Your query defines a new column, **`Balance`**, to calculate each employee's balance of vacation hours. ?

Which query will produce the report that you want?

```
a. proc sql;
   select name, spent,
          120-spent as calculated Balance
      from Company.Absences
      where balance <= 0;

b.  proc sql;
    select name, spent,
           120-spent as Balance
       from Company.Absences
       where calculated balance <= 0;

c.  proc sql;
    select name, spent,
           120-spent as Balance
       from Company.Absences
       where balance <= 0;

d.  proc sql;
    select name, spent,
           120-spent as calculated Balance
       from Company.Absences
       where calculated balance <= 0;
```

4. Consider this PROC SQL query: ?

```
proc sql;
   select flightnumber,
          count(*) as Flights,
          avg(boarded)
          label="Average Boarded"
          format=3.
      from sasuser.internationalflights
      group by flightnumber
      having avg(boarded) > 150;
```

The table *Sasuser.internationalflights* contains 201 rows, 7 unique values of **`FlightNumber`**, 115 unique values of **`Boarded`**, and 4 different flight numbers that have an average value of **`Boarded`** that is greater than 150. How many rows of output will the query generate?

a. 150

b. 7

c. 4

d. 1

5. You are writing a PROC SQL query that will display the names of all library cardholders who work as volunteers for the library, and the number of books that each volunteer currently has checked out. You will use one or both of the following tables: ?

- *Library.circulation* lists the name and contact information for all library cardholders, and the number of books that each cardholder currently has checked out.
- *Library.volunteers* lists the name and contact information for all library volunteers.

Assume that the values of **`Name`** are unique in both tables.

Which of the following PROC SQL queries will produce your report?

```
a.  proc sql;
    select name, checkedout
       from library.circulation
       where * in
          (select *
             from library.volunteers);

b. proc sql;
    select name, checkedout
       from library.circulation
       where name in
          (select name
             from library.volunteers);

c. proc sql;
   select name
      from library.volunteers
      where name, checkedout in
         (select name, checkedout
            from library.circulation);

d. proc sql;
   select name, checkedout
    from library.circulation
    where name in
        (select name
            from library.volunteers;);
```

6. By definition, a noncorrelated subquery is a nested query that ?

a. returns a single value to the outer query.

b. contains at least one summary function.

c. executes independently of the outer query.

d. requires only a single value to be passed to it by the outer query.

7. Which statement about the following PROC SQL query is *false?* ?

```
proc sql;
   validate
   select name label='Country',
          rate label='Literacy Rate'
```

```
      from world.literacy
      where 'Asia' =
         (select continent
            from world.continents
            where literacy.name =
                  continents.country)
      order by 2;
```

a. The query syntax is not valid.

b. The outer query must pass values to the subquery before the subquery can return values to the outer query.

c. PROC SQL will not execute this query when it is submitted.

d. After the query is submitted, the SAS log will indicate whether the query has valid syntax.

8. Consider the following PROC SQL query: ?

```
proc sql;
   select lastname, firstname,
          total, since
      from charity.donors
      where not exists
          (select lastname
             from charity.current
             where donors.lastname =
                   current.lastname);
```

The query references two tables:

- *Charity.Donors* lists name and contact information for all donors who have made contributions since the charity was founded. The table also contains these two columns: **`Total`**, which shows the total dollars given by each donor, and **`Since`**, which stores the first year in which each donor gave money.
- *Charity.Current* lists the names of all donors who have made contributions in the current year, and the total dollars each has given this year **`(YearTotal).`**

Assume that the values of **`LastName`** are unique in both tables.

The output of this query displays

a. all donors whose rows do not contain any missing values.

b. all donors who made a contribution in the current year.

c. all donors who did not make a contribution in the current year.

d. all donors whose current year's donation in *Charity.Current* has not yet been addedto **`Total`** in *Charity.Donors*.

9. Which statement about data remerging is true? ?

a. When PROC SQL remerges data, it combines data from two tables.

b. By using data remerging, PROC SQL can avoid making two passes through the data.

c. When PROC SQL remerges data, it displays a related message in the SAS log.

d. PROC SQL does not attempt to remerge data unless a subquery is used.

10. A public library has several categories of books. Each book in the library is assigned to only one category. The table *Library.Inventory* contains one row for each book in the library. The **`Checkouts`** column indicates the number of times that each book has been checked out. ?

You want to display only the categories that have an average circulation (number of checkouts) that is less than 2500. Does the following PROC SQL query produce the results that you want?

```
proc sql;
title 'Categories with Average Circulation';
```

```
title2 'Less Than 2500';
   select category,
          avg(checkouts) as AvgCheckouts
      from library.inventory
      having avg(checkouts) < 2500
      order by 1;
```

a. No. This query will not run because a HAVING clause cannot contain a summary function.

b. No. This query will not run because the HAVING clause must include the CALCULATED keyword before the summary function.

c. No. Because there is no GROUP BY clause, the HAVING clause treats the entire table as one group.

d. Yes.

Answers

1. Correct answer: d

To remove duplicate values from PROC SQL output, you specify the DISTINCT keyword before the column name in the SELECT clause.

2. Correct answer: d

To list rows that have no data (that is, missing data), you can use either of these other conditional operators: IS MISSING or IS NULL. The NOT EXISTS operator is used specifically with a subquery, and resolves to true if the subquery returns no values to the outer query.

3. Correct answer: b

When a WHERE clause references a new column that was defined in the SELECT clause, the WHERE clause must specify the keyword CALCULATED before the column name.

4. Correct answer: c

To determine how PROC SQL calculates and displays output from summary functions, consider the key factors. This PROC SQL query has a GROUP BY clause, and it does not specify any columns that are outside of summary functions. Therefore, PROC SQL calculates and displays the summary function for each group. There are 7 unique values of **`FlightNumber`**, but the HAVING clause specifies only the flights that have an average number of boarded passengers greater than 150. Because *4* of the 7 flight numbers meet this condition, the output will contain 4 rows.

5. Correct answer: b

Your PROC SQL query needs to use data from both tables. The outer query reads the name and number of books checked out from *Library.circulation*. The multiple-value noncorrelated subquery selects the names of volunteers from *Library.volunteers* and passes these names back to the outer query. The outer query then selects data for only the volunteers whose names match names returned by the subquery. The subquery is indented under the outer query's WHERE clause, is enclosed in parentheses, and does not require a semicolon inside the closing parenthesis.

6. Correct answer: c

A noncorrelated subquery is a nested query that executes independently of the outer query. The outer query passes no values to the subquery.

7. Correct answer: a

The syntax in this PROC SQL query is valid, so the first statement is false. The query contains a correlated subquery, so the second statement is true. The VALIDATE keyword is used after the PROC SQL statement, so the third statement is true. And the last statement correctly indicates that the VALIDATE keyword causes the SAS log to display a special message if the query syntax is valid, or standard error messages if the syntax is not valid.

8. Correct answer: c

In this PROC SQL query, the outer query uses the operator NOT EXISTS with a correlated subquery. The outer query selects all rows from *Charity.Donors* whose names do *not* appear in *Charity.Current*. In other words, this PROC SQL query output lists all donors who did *not* make a contribution in the current year.

9. Correct answer: c

The third statement about data remerging is correct.

10. Correct answer: c

PROC SQL can execute this query, but the query will not produce the results that you want. If you omit the GROUP BY clause in a query that contains a HAVING clause, then the HAVING clause and any summary functions treat the entire table as one group. Without a GROUP BY clause, the HAVING clause in this example calculates the average circulation for the table as a whole (all books in the library), not for each group (each category of books). The output contains either all the rows in the table (if the average circulation for the entire table is less than 2500) or none of the rows in the table (if the average circulation for the entire table is greater than 2500).